Cordova - Guide - Extras & Options

- Dr Nick Hayward

A brief overview of extra options and features for Cordova.

Contents

offline ready apps

We'll now work our way through a few additional considerations as you prepare your app for users, testing, and publication.

We'll start with a consideration of offline support for our mobile app.

As noted before, a mobile app needs to consider usage with limited or no network connectivity. This might be due to poor network reception, or an explicit act by the user to restrict data usage.

With Cordova based apps, we can, of course, allow for many situations where network access is restricted or unavailable. Online web apps, which need to rely on local caches, or the much maligned **AppCache** API, Cordova bundles the required files as it compiles the app.

However, there are still many considerations for effective offline usage. We need to consider many disparate parts of our app that will be affected by offline usage. For example, it's not just an issue with loss of connectivity to services, data, collaborative features &c. It is also an issue with the UI design, interaction, and features.

For example, if a user is able to click a button, select an option whilst online, what happens if they are now offline? Naturally, they will be unable to access the end service for the request. However, as designers and developers we should be proactive in removing this option whilst offline.

For example, if a user is able to click a button, select an option whilst online, what happens if they are now offline?

Naturally, they will be unable to access the end service for the request. However, as designers and developers we should be proactive in removing this option whilst offline. So, we may simply remove this button and option if network connectivity is lost, or more commonly update the element's state to `inactive` or effectively ghost-out the appearance and interaction of the element.

This act of updating the state of an element for offline usage has a number of benefits. With a disabled state, not only is the visual rendering updated, event listeners should also become inactive. So, we remove any potential issues and errors with the logic of the app due to the loss of connectivity.

We should, of course, also offer feedback to the user to inform them why an element, option, or interaction is no longer available. We can simply inform them of the state of the network, for example.

With a Cordova app, as it loads we can set a listener for network related events. We can then continue to check and monitor the status of the network as the app is running, and then trigger changes in state as required during the lifecycle of our app.

So, our app will be able to respond accordingly simply by checking whether it's online or offline. Naturally, we need to monitor the state of the app as a user may switch between states of network coverage and usage.

Cordova provides the useful **Network Information** plugin to help us monitor the state of our app's network connection. This plugin has two notable features. Firstly, we can use this plugin to monitor the type of connection our device is currently using. It will return a number of different possible types, including `unknown`, `offline`, or one of the available network options. These might include WiFi, 4G, 3G &c.

Secondly, we can then respond to events within our app for `offline` and `online`. Of course, by simply listening to these events, we'll now be able to update and modify our app correctly according to a given state. We can also use these events to offer the necessary feedback to our users.

To use this network functionality with an app, we'll need to add the **Network Information** plugin,

```
cordova plugin add cordova-plugin-network-information --save
```

Once we've installed and tested this plugin, we can then check the standard `navigator` object for the connection type of our device. This will help us determine whether the user's current connection is WiFi, 4G, &c. Then, by monitoring this connection type we can update our app's UI, interaction, and logic.

So, we can start by adding the necessary listeners for the network state of our app,

```
document.addEventListener("offline", offlineState, false);
document.addEventListener("online", onlineState, false);
```

These event listeners simply allow us to respond accordingly to a change in the monitored status of an app's network connection. As the app loses or regains network connectivity, we can use these listeners to maintain a correct state for our app.

We can use these custom functions, `offlineState` and `onlineState`, to update our app's UI for a disabled or enabled state, and then offer feedback to the user.

```
//handle offline network state
function offlineState() {
  //handle offline network state
        console.log("app is now offline");
        //show ons alert dialog...
        ons.notification.alert('your app is now offline...');
}
```

```
//handle online network state
function onlineState() {
// Handle the online event
  var networkState = navigator.connection.type;
        console.log('Connection type: ' + networkState);
  if (networkState !== Connection.NONE) {
    //use connection state to update app, save data &c.
  }
  ons.notification.alert('Connection type: ' + networkState);
}
```

The Cordova docs for this plugin also suggest the following useful function to help quickly monitor and check our app's network status,

```
function checkConnection() {
  var networkState = navigator.connection.type;
        console.log('check connection requested...');
  var states = {};
  states[Connection.UNKNOWN]  = 'Unknown connection';
  states[Connection.ETHERNET] = 'Ethernet connection';
```

```
    states[Connection.WIFI]     = 'WiFi connection';
    states[Connection.CELL_2G]  = 'Cell 2G connection';
    states[Connection.CELL_3G]  = 'Cell 3G connection';
    states[Connection.CELL_4G]  = 'Cell 4G connection';
    states[Connection.CELL]     = 'Cell generic connection';
    states[Connection.NONE]     = 'No network connection';

        console.log('Connection type: ' + states[networkState]);
}
```

Image - Network Information - part 1

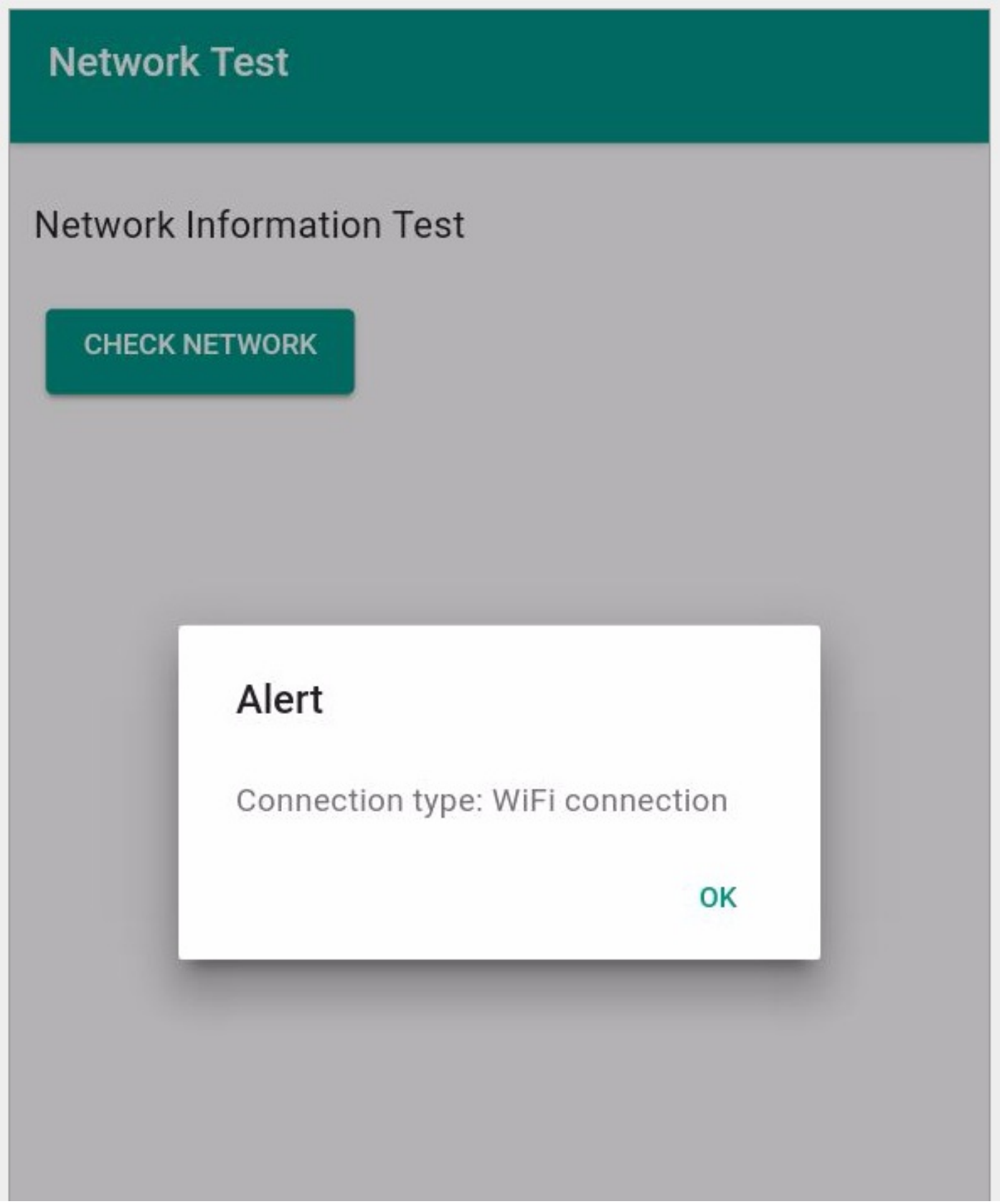


Image - Network Information - part 2

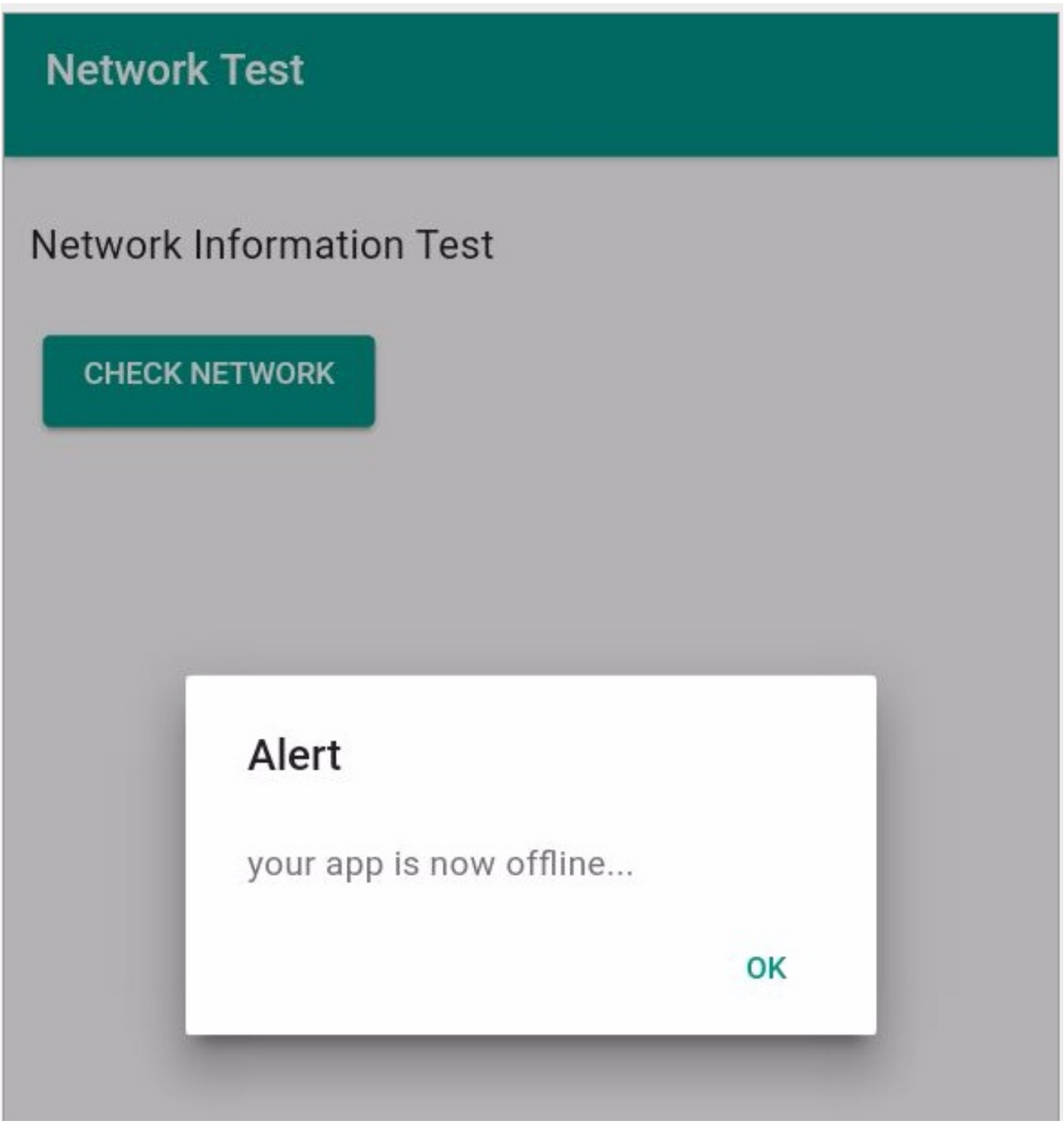
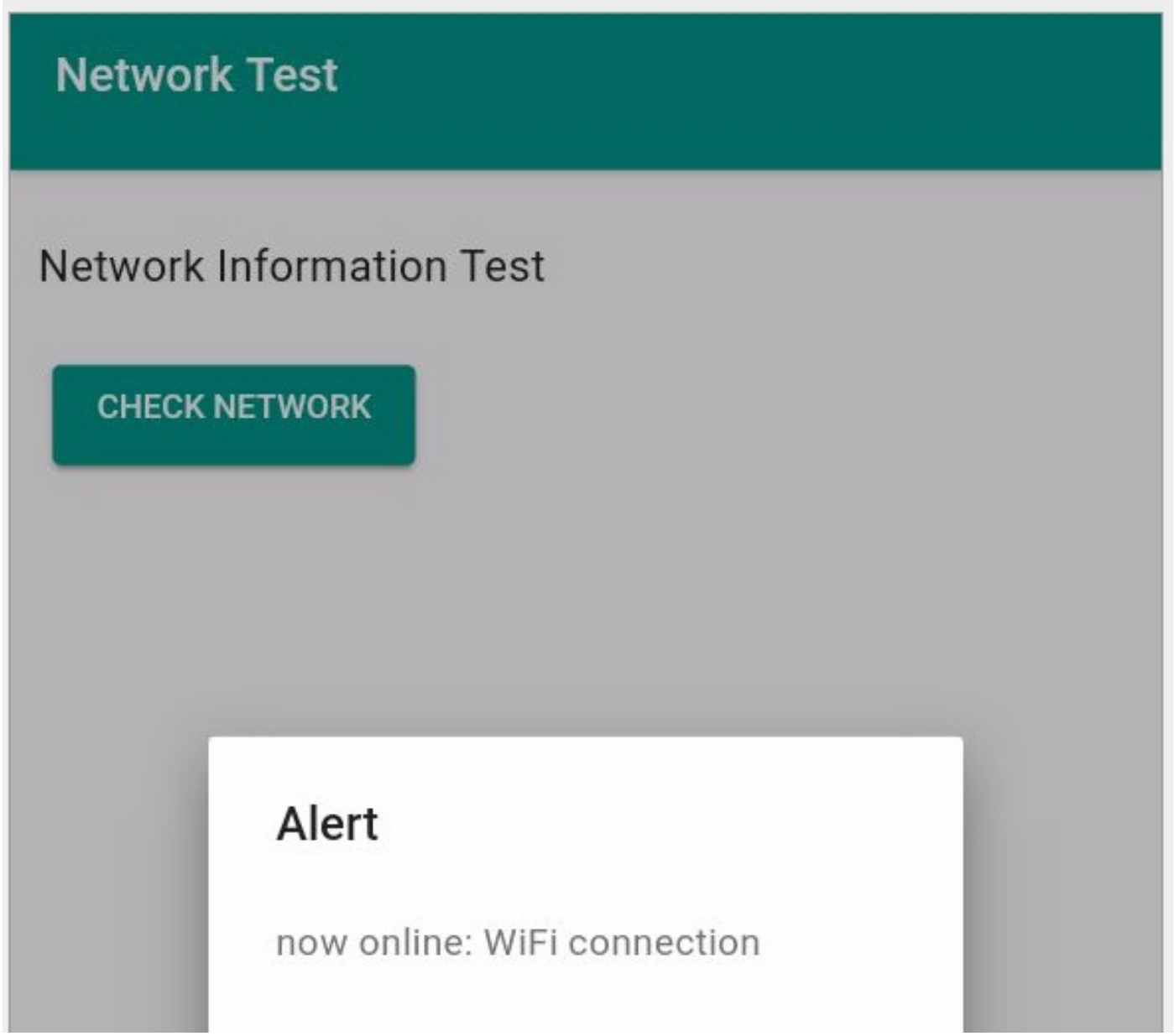
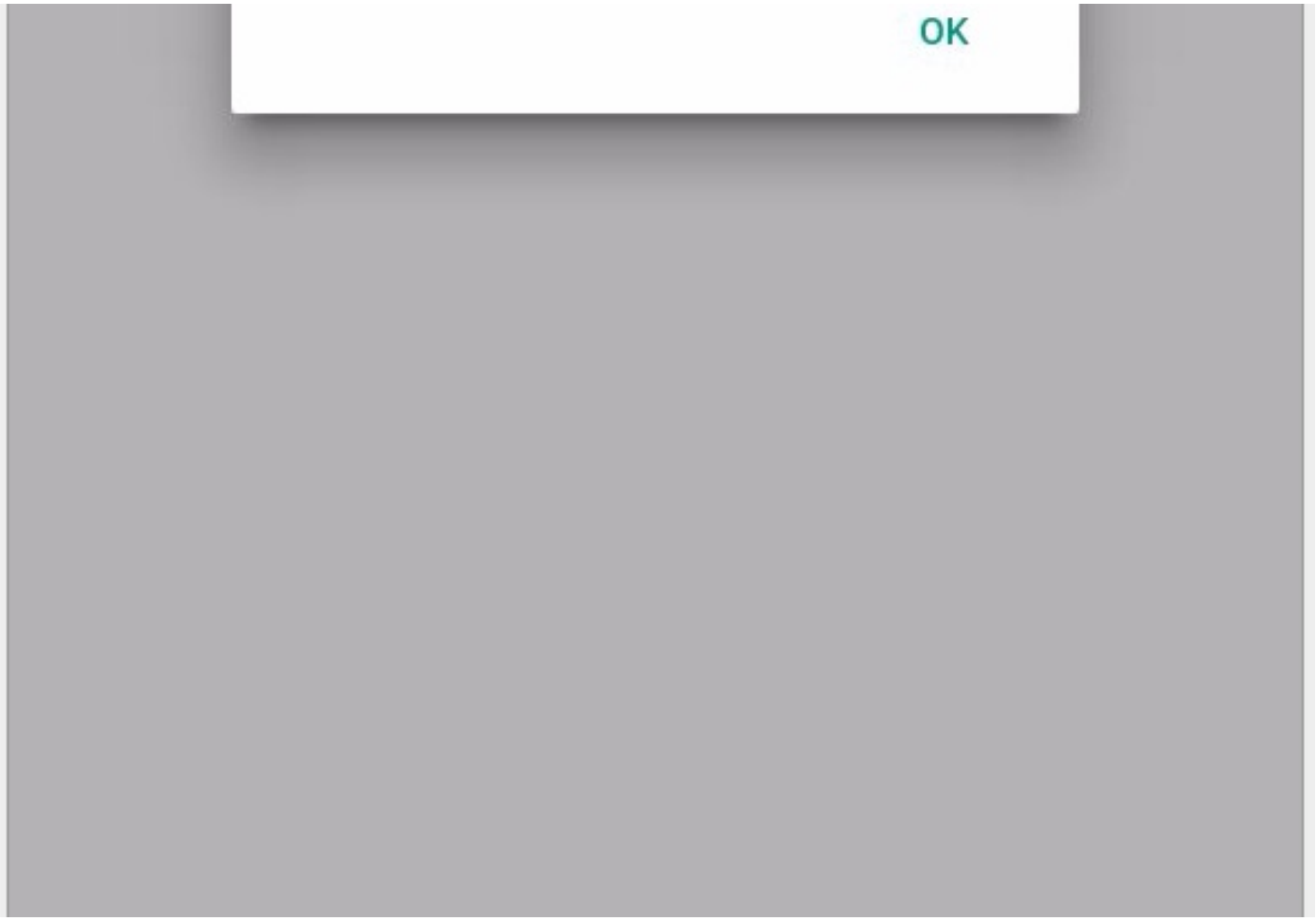


Image - Network Information - part 3





add International support

As we start to consider publication and release for a mobile app, we need to remember that our app will be used by many different people, in different countries with varying standards, metrics, timezones, measurement systems, and so on. In effect, we need to ensure that input and output for our app is correctly defined and structured to meet such international requirements.

So, we can now consider and test globalisation for our app. Cordova, again, provides a very useful **Globalization** plugin for such requirements. We can add this to our projects using the standard CLI command,

```
cordova plugin add cordova-plugin-globalization
```

This plugin uses a device's settings to determine and monitor a user's defined **locale**, **language**, and **timezone**. So, we might have a user with a defined locale of **USA** but a language setting of **UK English**. This means the OS and supported apps will output dates, numbers, measures &c. in a USA compliant format, but render the language itself using UK English. Certainly a strange hybrid, but something that the user has specified. Therefore, we need to ensure that our app adheres to this preference.

As you might expect by now, we can use this plugin with the defined global object,

```
navigator.globalization
```

once the standard `deviceready` event has returned successfully. So, we can start by checking a user's defined language for the current app,

```
navigator.globalization.getPreferredLanguage (
  //set success and error callbacks...
  function(language) {
    console.log('language = '+language.value);
  }, function() {
```

```
    console.log('error with language check...');
  }
);
```

We can also check a user's defined locale, which follows the same general pattern to the language check.

```
navigator.globalization.getLocaleName (
  //set success and error callbacks...
  function(locale) {
    console.log('locale = '+locale.value);
  }, function() {
    console.log('error with locale check...');
  }
);
```

We can also update and customise our app's dates and times to correctly match the specified locale settings. For this, we can use the `dateToString()` method with the navigator object,

```
navigator.globalization.dateToString(
    new Date(),
    function (date) { alert('date: ' + date.value + '\n'); },
    function () { alert('Error getting dateString\n'); },
    { formatLength: 'short', selector: 'date and time' }
);
```

In this example from the Cordova API docs, we can see that the date is created using JavaScript's `Date()` constructor. We can then use it with the `dateToString()` method on the navigator object to ensure the rendered date is formatted correctly to match the set locale.

Image - International Support - part 1

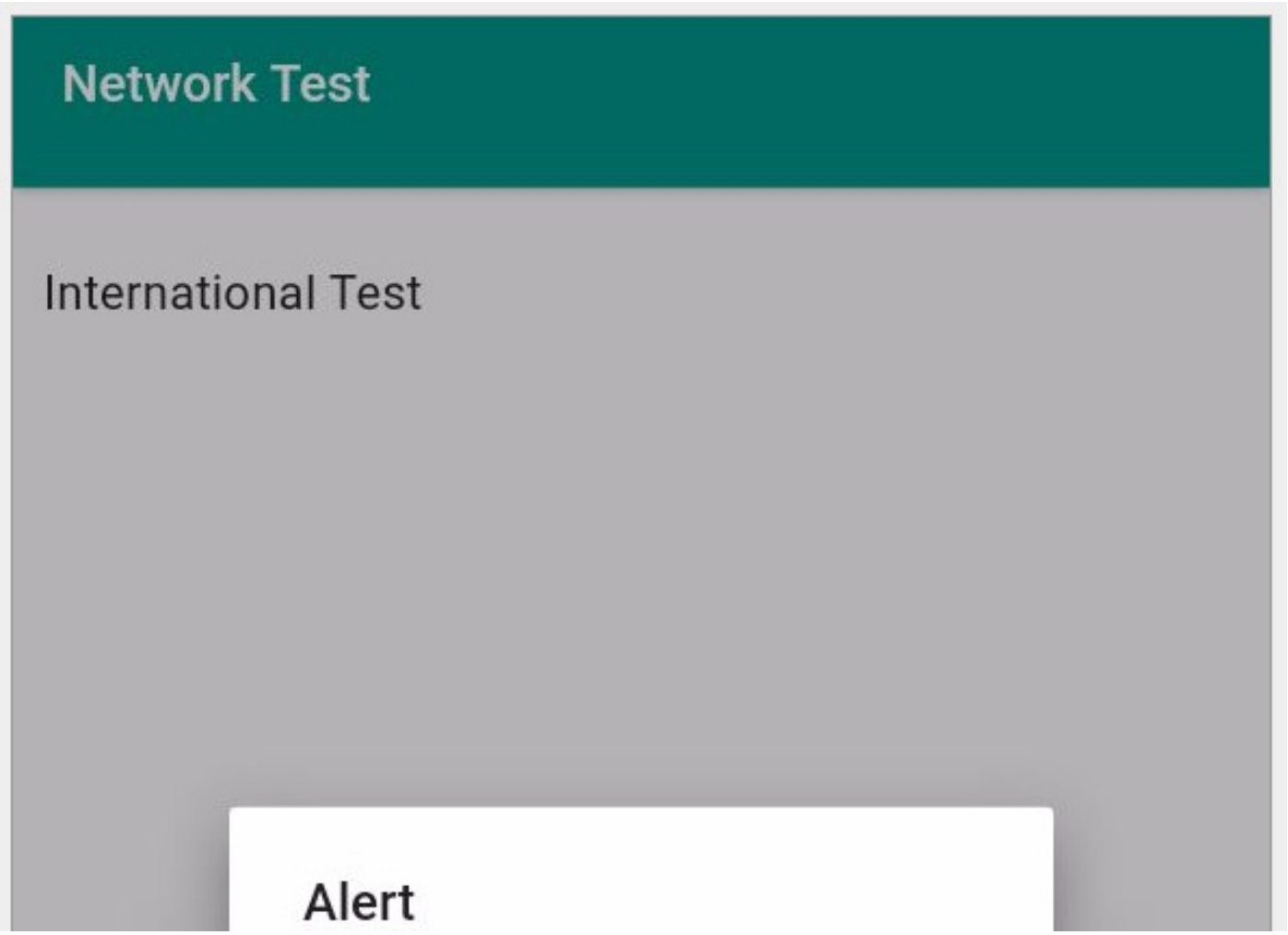
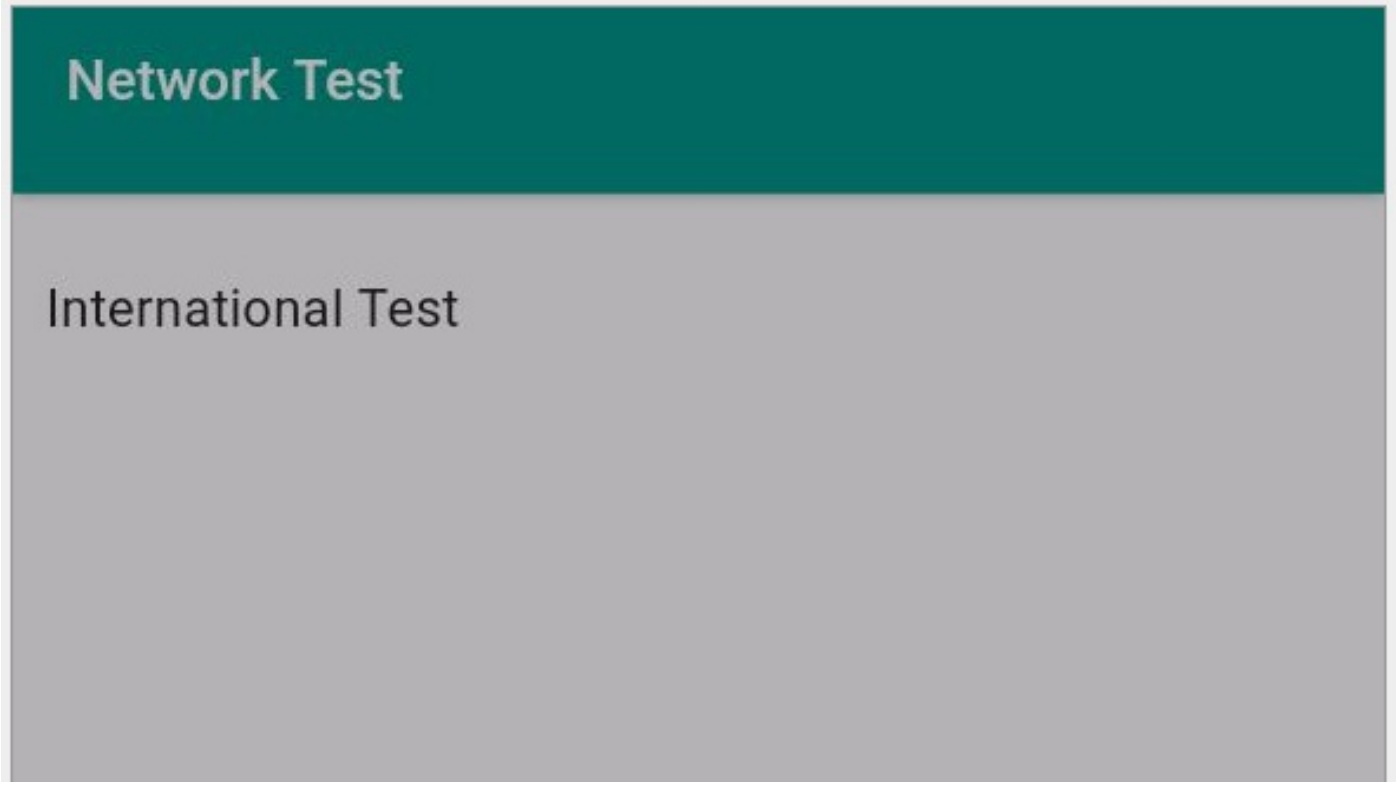


Image - International Support - part 2



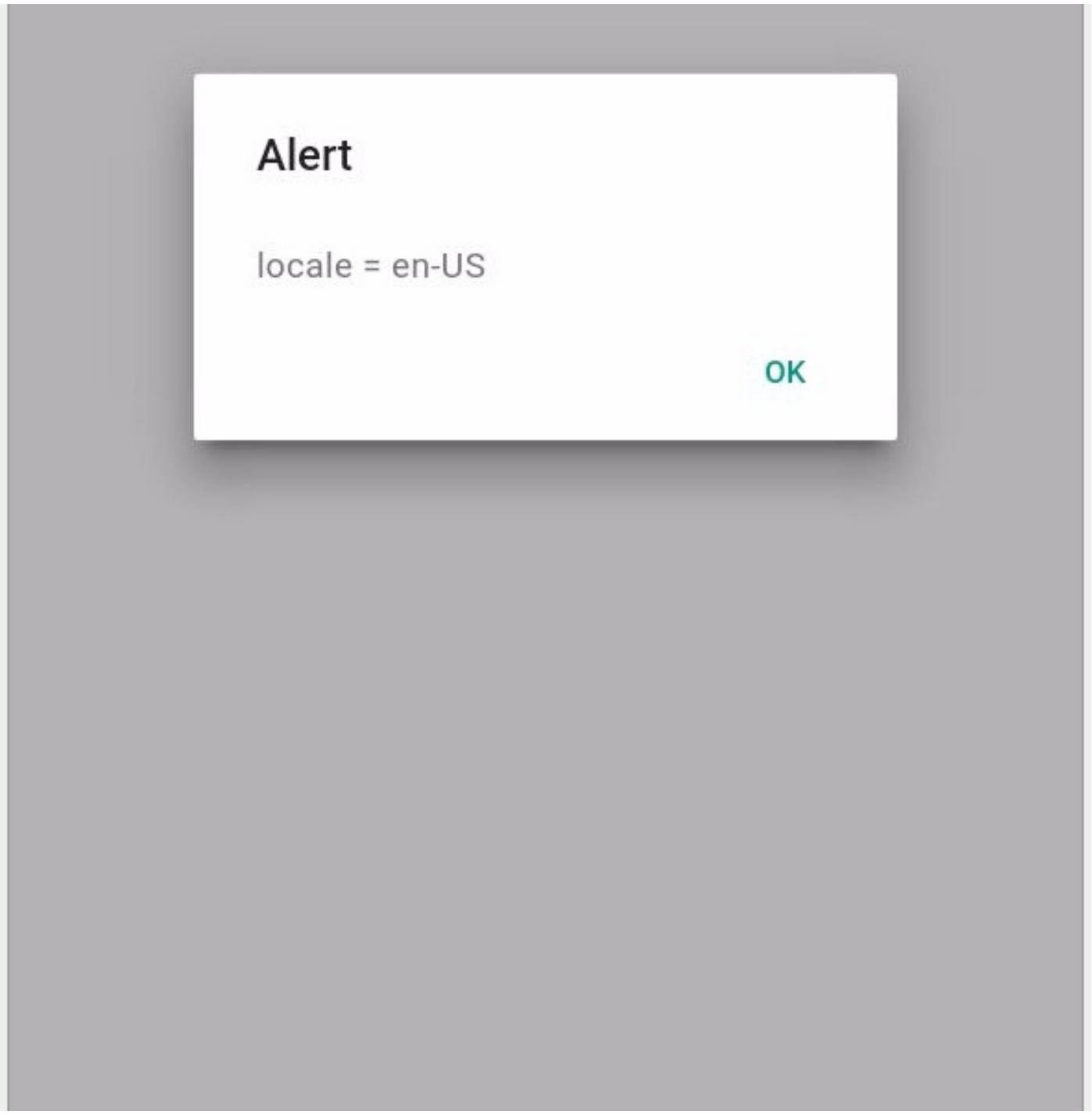
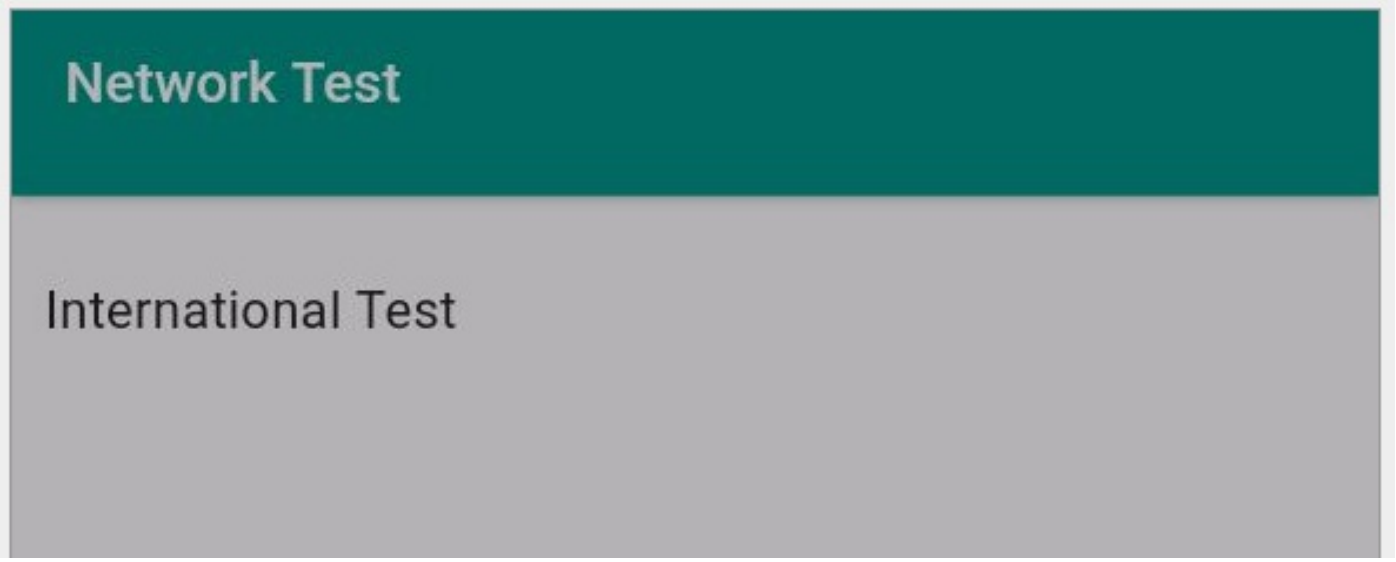
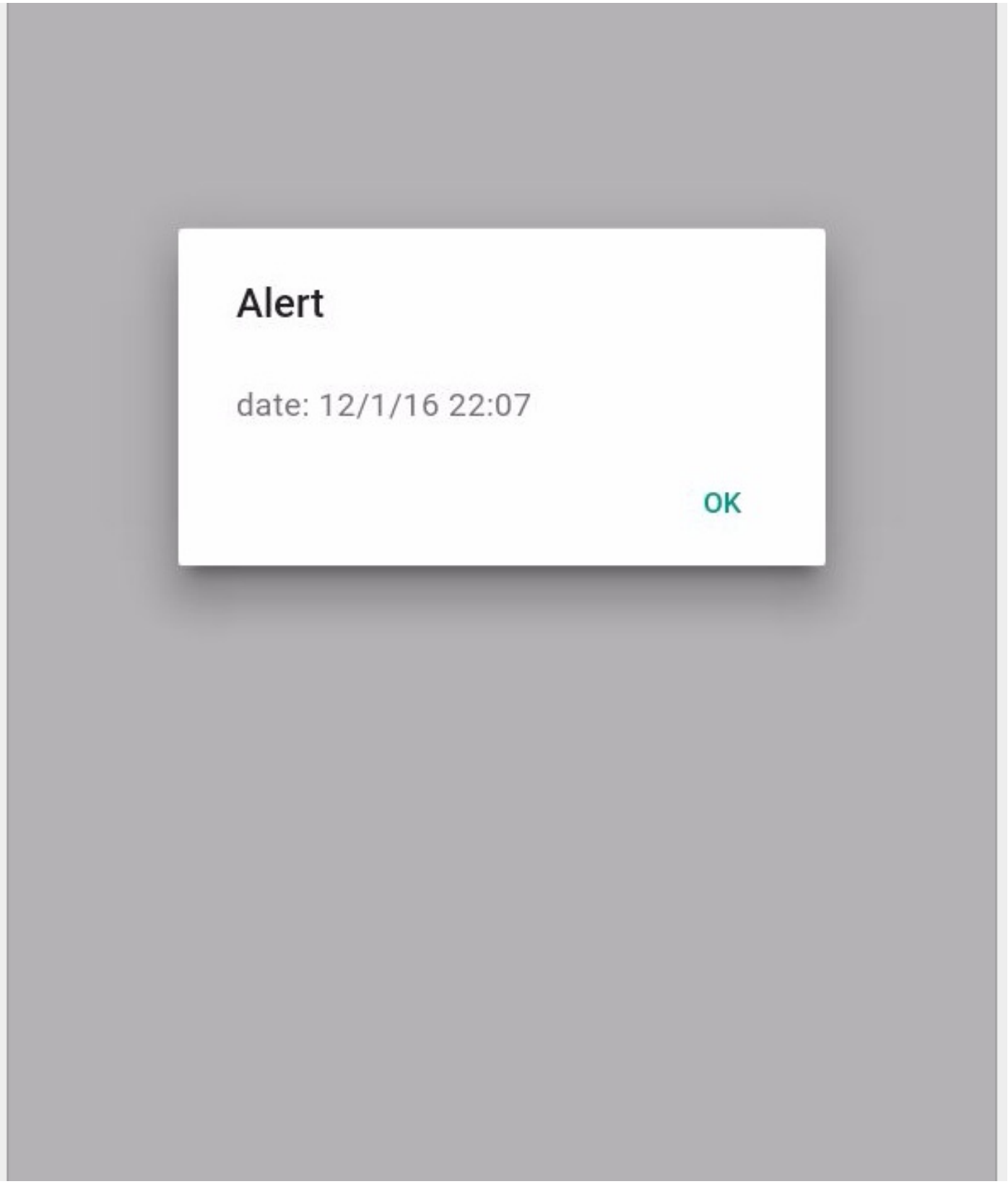


Image - International Support - part 3





build and customisation - config.xml

We've already seen the `config.xml` file, but there are a few additional preferences that we can consider in the metadata. We can modify the values of these preferences to help us configure and setup our app with greater precision and customisation.

So, Cordova uses this file to help define various aspects and structures within an app. It's used as part of the build process, and to help us defines the underlying metadata and settings for running our initial app.

In addition to the standard metadata for author, description, app name, and ID, there are also a few additional, useful preferences. These include specifying the default start file as the app loads, a security setting for resource access, and a minimum API for building the app.

When we create a Cordova app using the CLI tool, the default start file will be specified as `index.html`. However, as necessary within the structure of our custom app, we can also update this value to a different file,

```
<content src="custom.html" />
```

Within a standard Cordova structure, it's unlikely you would need to update this value, but at least you know it's possible.

We can also update our app's settings to define access privileges and domains for remote resources. For example, any CSS stylesheets, JavaScript files, images, remote APIs, and servers. These are specifically remote resources that are not bundled with the app itself.

Cordova refers to this setting as a **whitelist**. It's now been moved to a specific plugin, which is added by default as we create an app.

The default value for this setting is global access, e.g.

```
<access origin="*" />
```

For many apps, this setting will be fine.

However, we may need to restrict access due to user input in our app, remote loading of data, and so on. In effect, we might consider restricting our app to specific domains.

We can add as many `<access>` tags as necessary for our app, e.g.

```
<access origin="http://www.test.com" />
<access origin="https://www.test.com" />
```

This allows our app to access anything on this domain, including secure and non-secure requests.

We can also add subdomains relative to a given domain, simply by prepending a wildcard option to a specified domain,

```
<access origin="http://*.test.com" />
<access origin="https://*.test.com" />
```

So, if we know that our app requires access to specific remote domains, we can update our app's settings to ensure access is restricted correctly.

Beyond updating the initial default `config.xml` setting file, we can also add further metadata and preferences to help customise our app.

For example, we've already seen how to add custom icons and splashscreens for our app. In addition, we can also add further settings for plugins, specific installed and supported platforms, general preferences for all platforms, or restrict to a single platform.

As an example, for general preferences there are five global options we might consider for our app's settings. These include

- BackgroundColor
  - Android and iOS support for a specific fixed background colour
- DisallowOverscroll
  - Android and iOS support to prevent a rendered app from moving off the screen
- Fullscreen
  - Android (but not iOS) support to determine whether an app should cover the whole screen or not
  - e.g. useful for kiosk style apps...

- HideKeyboardFromAccessoryBar
  - iOS (but not Android) support for hiding an additional toolbar above a keyboard
- Orientation
  - Android (but not iOS) support for locking an app's orientation

We can add any necessary preferences as follows simply by using the `<preference>` element in our `config.xml` file. For example,

```
<preference name="fullscreen" value="true" />
```

We can add as many preferences as necessary for our app's configuration. We can also customise our preferences for a specific platform, for example restricting a preference to just Android or iOS. For example,

```
<platform name="android">
  <preference name="DisallowOverscroll" value="true" />
</platform>
```

build and customisation - merge options

For many apps we build using Cordova, we'll be able to develop using a single code base with platform specific preferences and UI customisations. This is the build pattern we've been using for our test apps.

However, we may encounter scenarios where we prefer to create a distinction in the app's design or functionality. One way to achieve this separation between platform specific code is to use a Cordova pattern known as **merges**.

We need to create a new folder called `merges` in our app's root directory. This is the root directory for the app itself, and not the `www` directory.

We can use this new `merges` folder to help us develop platform specific requirements, including stylesheets. We can add a sub-directory within `merges` for each platform we want to support in our app. As we build our app using Cordova's CLI tool, if there is a `merges` folder, it will be checked for each platform. If a required file exists, it will then replace the equivalent file in the app's `www` directory. If there is a file in a plaform sub-directory of the `merges` folder that does not exist in the `www` directory, then it will simply be added as the app is built.

For example, our app's updated structure will be as follows

```
config.xml
|-- hooks
|-- merges
    |__ android
    |__ ios
|-- platforms
|-- plugins
|-- www
```

An example usage might include specific stylesheets per platform.

For example, in our app's `index.html` file we can add a link reference to a stylesheet. The file for this stylesheet is added as usual to our app's `www` directory, but we can now leave this file blank for the overall project. We then add a matching CSS file to each platform directory in the `merges` folder. This CSS file will then be added to our platform specific app as it is built by Cordova. For example, our directory structure can be updated as follows

```
config.xml
|-- hooks
|-- merges
    |__ android
```

```
        |__ css
            |__ platform.css
    |__ ios
|-- platforms
|-- plugins
|-- www
    |__ css
        |__ platform.css
    |__ ...
```

This allows us to add specific styling, layout, and design requirements for each supported platform quickly and easily.

build options - hooks

We've been using Cordova's CLI tool to help create our apps, add required platforms and plugins, build our apps, and so on. However, so far we've been using default options and patterns.

With **Hooks**, we can customise the behaviour of Cordova's CLI relative to a given project. In effect, **Hooks** are scripts that are able to interact with the CLI tool for a given command and action. We can often consider these **Hooks** in two distinct scenarios, before and after an action is executed by the CLI tool.

For a standard Cordova CLI command, we might consider adding a **hook** before or after that command and action is called and executed. Many uses of **hooks** include automation of standard build options, tools, and commands. For example, we might need to add the same plugins for a series of apps we're building and testing. We can create a **hook** that adds a list of plugins after adding a platform to a project.

The Cordova CLI tool will, by default, check for **hook** scripts in the `hooks` directory. To add a **hook**, we simply create a sub-directory in the `hooks` directory with the same name as a defined **hook**. Cordova will then check this sub-directory for scripts to execute. It's also important to note that scripts will be executed in alphabetical order by filename.

Another interesting option with **hooks** is that they can be written in any language supported by the host computer. In effect, as long as the script can be run from the command line, it can used with the Cordova CLI tool. However, many **hooks** are still written in JavaScript, which is then executed using Node.js.

prepare for release

As we finalise our Cordova app, we need to consider how to correctly prepare and package our application for publication to one or more of the available app stores. However, each of the major app stores tends to require that we follow a similar, prescribed pattern for preparation and publication of an app.

To prepare our app for publication, we can begin by transitioning from an initial development version of our app to a, hopefully, stable release version. This usually requires an application that has been signed by the developer with a password. Effectively, we are defining ownership of the app, and accepting responsibility for its publication, contents, and so on.

Next, of course, we need to submit our app to a store for publication. In addition to preparing the app itself, we will normally be required to provide descriptions for the app itself, and provide a minimum of screenshots for general usage and prominent features. In effect, we can use this supplementary information as our way of trying to sell our app.

prepare for release - Play Store

Releasing an Android app is considerably less involved than its iOS counterpart. Within reason, developers can release and publish a vast array of application types.

So, let's begin with Android and publication of our app on the Play Store. To help us conceptualise this overall process, we may consider it as a division between preparation of the app, and then publication.

For the initial preparation, we begin by signing our app with a key, which we can create from the command line. Then, we can use Cordova build tools to create a final, release build of our application.

With a prepared, publication ready Cordova app, we can upload our app to Google's Play Store for publication. As noted, we'll need to provide some additional supporting information, including a title for our app, icons, description, and so on. We can then mark our app as published.

prepare for release - signing

To prepare our app for a store, we need to sign it using a key store and key prior to publication. The key is the signature, which is saved in the defined key store.

We need to be particularly careful with this key, of course, as it provides security and authentication for the publication of our app. Therefore, take care with the key file itself, and the password.

We can sign our app using a command line tool provided by Java, `keytool`. An example keytool command for our app might be as follows,

```
keytool -genkey -v -keystore my-app-ks.keystore -alias my-app-ks -keyalg RSA -keysize 2048 -validity 10000
```

With this command, we are creating both the keystore and the required key for our app. There are a few arguments that we need to consider for this command,

- `my-app-ks.keystore` - this is the filename for the keystore, which can be set to a preferred name for your app.
- `my-app-ks` - this is the name of the alias for the keystore. Again, a developer can specify their preferred name, which can be a simple, plain text name for the keystore.

After submitting this command, **keytool** will ask a number of question, which help setup the key for your app. Simply answer these questions, and your app's keystore and key will be created.

Image - Keytool - Create a Keystore